# RDFMatView: Indexing RDF Data for SPARQL Queries

Roger Castillo, Christian Rothe, and Ulf Leser

Humboldt Universtiy of Berlin
{castillo,rothe,leser}@informatik.hu-berlin.de
http://www.hu-berlin.de/

**Abstract.** The *Semantic Web* as an evolution of the *World Wide Web* aims to create a universal medium for the exchange of semantically described data. The idea of representing this information by means of directed labelled graphs, *RDF*, has been widely accepted by the scientific community. However querying *RDF* data sets to find the desired information often is highly time consuming due to the number of comparisons that are needed. In this article we propose indexes on *RDF* to reduce the search space and the *SPARQL* query processing time. Our approach is based on materialized queries, i.e., precomputed query patterns and their occurrences in the data sets. We provide a formal definition of RDFMatView indexes for SPARQL queries, a cost model to evaluate their potential impact on query performance, and a rewriting algorithm to use indexes in SPARQL queries. We also develop and compare different approaches to integrate such indexes into an existing SPARQL query engine. Our preliminary results show that our approach can drastically decrease the query processing time in comparison to conventional query processing.

**Key words:** SPARQL, Indexing, RDF, Materialized views

# 1 Introduction

The *Semantic Web* as an evolution of the *World Wide Web* is an important initiative which has recently gained momentum. It aims to create a universal medium for the exchange of data where data can be shared and processed by automated tools as well as by people. The basis for this proposal is a logical data model called Resource Description Framework (RDF) [1]. An RDF data set is a collection of statements called *triples*, of the form *(s,p,o)* where *s* is a subject, *p* is a predicate and *o* is an object. Each triple states the relation between subject and object. A set of triples can be represented as a directed graph where subjects and objects represent nodes and predicates represent edges connecting these nodes. The SPARQL query language is the official standard for searching over RDF repositories [2]. It supports operations of triple patterns, similar to select-project-join queries in relational databases. For instance, using the SPARQL query in Listing 1 we can retrieve articles written by Albert Einstein from a local RDF data store.

```
SELECT ?title WHERE {
?article <hasTitle> ?title.
?article <hasAuthor> ?author.
?author <hasName> "Albert Einstein".
}
```

**Listing 1.** SPARQL query to retrieve all articles written by Albert Einstein

Listing 1 illustrates a simple SPARQL query where each join is denoted by a dot. The whole query pattern can be seen as a graph pattern that needs to be matched in the RDF data set. Predicates may also contain, which increases the complexity of query evaluation.

The increasing amount of RDF data on the Web requires the development of approaches for efficient RDF data management. Almost all recent implementations of SPARQL are build upon relational databases (e.g. Jena [3], 3Store [4, 5], or Sesame [6]). In these systems, a SPARQL query is translated into one or more SQL queries over a relational representation of the underlying RDF data set. This relational representation usually stores RDF triples in one or a few tables. Consequently, answering a SPARQL query consisting of more than one pattern requires the computation of roughly as many joins as the query has patterns. Optimizing these joins is one of the critical issues to obtain scalable SPARQL systems.
The typical architecture used by these approaches is shown in Figure 1. This architecture is based on a 3-Layer schema which consists of the SPARQL query interface, the RDF data representation and the underlying database.

In this work, we propose the use of materialized SPARQL queries to speed-up queries. We target large data sets and SPARQL queries consisting of many basic graph patterns. Examples of huge data sets are, for instance, the UniProt database containing more than 600 million triples [7] or the W3C SWEO Linking Open Data Community with more than 4 billion triples [8]. With such datasets, executing a query with many graph patterns becomes a problem.

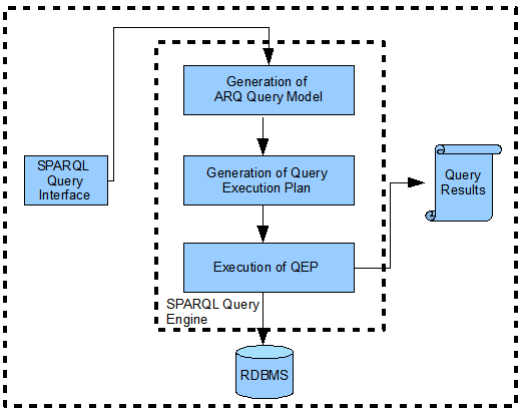



**Fig. 1.** SPARQL Query Processing using a typical RDBMS - Backend Architecture.

```
SELECT * WHERE {
?s1 ?p1 ?o1 .
?o1 bif:contains "hexokinase" .
?s1 <type> ?type1 .
?s1 <comment> ?comment1 .
?s1 <label> ?label1 .
?s1 <isA> ?s2 .
?s2 <comment> ?comment2 .
?s2 <label> ?label2 .}
```

**Listing 2.** Example SPARQL query. Information about Hexokinase enzyme [9]

Consider the query given in Listing 2, which a real-life query taken from [9] gathering information about a certain enzyme from a database. Executing this query on a conventional SPARQL processor will result in the computation of seven-way self-join of the triple table. However, one can safely assume that the types, labels, and comments of an object are used together very often. Therefore, we suggest to materialize this information by executing a query off-line and storing the results inside the system. Given this materialized query, the query could be computed with only four joins.

In contrary to previous approaches which focus on indexing the relational representation of an RDF storage scheme, we fully exploit the RDF graph-structure for indexing. We are not indexing single attributes or triples, but fractions of queries that occur frequently in an expected workload. Therefore, our approach is a *native RDF/SPARQL indexing* method whose concepts are viable for all possible implementations of RDF stores. Such an approach requires solutions to a whole series of problems. First, queries have to be materialized. Second, the query must be analyzed to identify all materialized queries that could be used, and especially to identify the best such query or best combination of such queries that should be used. This requires complex query planning algorithms and a cost model. Third, the query processing itself must be modified to be

able to retrieve materialized results and to combine them with those parts of the original query that are not covered by the indexes.

More formally, this paper presents solutions to the following problems.

- Generation of execution plans to cover a query. Given a query $Q$ for a data graph $G$ and the set $I$ of all indexes on $G$, the first step is to define which indexes are usable for speeding up $Q$. To this end, we need to generate all possible mappings between the index pattern and the query.
- Definition of a cost model to assess different query execution plans. Each plan differs from the others according to the parts of the query that are covered by indexes, which in turn leads to different sizes of intermediate results and different parts of the query that need to be executed and combined with the materialized parts. The objective of the cost model is to assess all possible plans and to find those with minimal estimated cost.
- Rewriting of a SPARQL query to substitute covered query patterns by RDFMatView indexes. When a combination of RDFMatView indexes is selected, its materialized results must combined to occurrences of the covered query pattern. There are two different cases how this may happen:
  1. The combination of indexes completely cover the query pattern. Thus, the solution to the query can be generated completely by joining the indexes.
  2. The combination of indexes only partially cover the query pattern. Then, the query solution needs to be generated by joining the results of the chosen indexes with the residual parts of the query pattern.

In this report, we present solutions to all of these problems. We first discuss related work in Section 2. Section 3 introduces basic concepts of RDF and SPARQL. Section 4 presents the fundamental principles of *RDFMatViews* and a strategy to use them for SPARQL query processing. Section 5 introduces the cost model used to evaluate different query plans. We describe different ways to introduce materialized queries into an existing SPARQL processor in 6. Finally, we give an evaluation of our method in Section 7 and conclude in Section 8.

We presented a theoretical framework for using materialized SPARQL queries as indexes in [10]. In the present work, we show the practical applicability of this framework by describing and comparing several ways to integrate materialized views into an existing SPARQL query processor and by providing an evaluation on the speed-ups that can be achieved using our methods.

Clearly, in a setting such as ours it is also important to provide algorithms to keep materialized queries up-to-date, and to give a user hints on which queries should be materialized to best (best in terms of space/cost-efficiency) support a given workload. However, these questions are out-of-scope of our current research.

## 2 Related work

In this section, we discuss published techniques for indexing SPARQL queries and contrast them to our work.

Most works focus on optimizing a single join. In [11] Abadi et al. propose a vertical partition approach for Semantic Web data management. An enhancement of this approach is proposed by Weiss et al. in [12]. Therein, RDF data is indexed in six possible ways, i.e., an index for each possible ordering of the three RDF elements. Each instance of an RDF element is associated with two vectors; each such vector gathers elements of one of the other types, along with lists of the third-type resources attached to each vector element. This scheme is capable of speeding up single joins tremendously, but storage requirements are very high, which becomes a serious issue when using huge data sets. Neumann and Weikum developed *RDF-3X*, a SPARQL engine implementation pursuing a RISC-style architecture – a streamlined architecture – with specific-designed data structures and operations [13]. The authors overcome the "giant-triples-table" [11] bottleneck by creating a set of indexes and a fast way pf processing merge joins. Similar to [12], RDF-3X maintains all six possible permutations of subject, predicate and object in six separate indexes. The authors also present a compression algorithm to alleviate the problem of space consumption.

All these approaches have in common that they focus on single joins. When faced with queries consisting of multiple basic graph patterns, they still have to compute multiple joins (although every single join is faster). In contrast, our work specifically targets the speed-up of complex queries consisting of many basic graph patterns.

There is also some other work that considers groups of patterns. In [14] the authors present *GRIN*, a lightweight indexing mechanism for RDF data. The main idea is to draw circles around selected *center* vertices in the graph where the circle would comprise those vertices in the graph that are within a given distance of the "center" vertex. Basically, GRIN is a binary tree where the set of leaf nodes form a partition of the set of triples in the RDF graph. An interior node implicitly represents the set of all vertices in the RDF graph that are within a specific unit of distance. To evaluate a query, GRIN derives a set of inequality constraints from the query. These constraints are evaluated against the nodes of the GRIN index. A similar indexing approach is presented in [15]. This work proposes to create a set of indexes of precomputed joins using all possible join combinations between triple patterns. As [14], this approach aims to create a general purpose set of indexes based on joined triple patterns, but the number of indexes to manage is not practical when the number of joined triples is >= 3.

The two systems just described index groups of and not just single triples. However, they propose to apply their techniques to all RDF triples, while we only build user-chosen indexes. Our work fundamentally is based on the assumption that some patterns are used together in queries more frequently than others, and that indexing those combinations suffices to gain large speed-ups at manageable space and maintenance cost.

One can best describe the differences between our ideas and that of other RDF indexing schemes by drawing a parallel to B*-indexes in relational databases [16]. Nobody would sensibly suggest to speed up queries by indexing every attribute; instead, systems assume that developers have a rough idea about the types of queries that need

to be answered and therefore index only the relevant attributes. Furthermore, optimal speed-up can only be achieved when also combinations of attributes can be indexed, and not only single attributes. In this sense, the former approaches index every single attribute, the latter indexes every possible combination of attributes, and we suggest to index only selected combinations of attributes.

## 3 Preliminaries: RDF and SparQL

The SparQL query language has increased its popularity since it recently reached a candidate recommendation of the W3C [2] for retrieving information from RDF [1] graphs stored in semantic storage systems. We make the assumption that our audience is familiar with RDF and SparQL. Thus, we only describe the most important terms from this specification which are required for our project. For formal definition of these concepts we refer the reader to the RDF specification [1].
An RDF graph is a set of RDF Triples, which in turn consist of RDF Terms. An RDF Term consists of an IRI,[1] blank node or an RDF Literal. The set of all RDF terms is denoted in this article as *RDF-T*. The building blocks of a SparQL query are triple patterns. Triple patterns are basically RDF triples that can contain variables. A basic graph pattern is made of a set of triple patterns. In the rest of this paper we will refer to basic graph pattern as pattern.

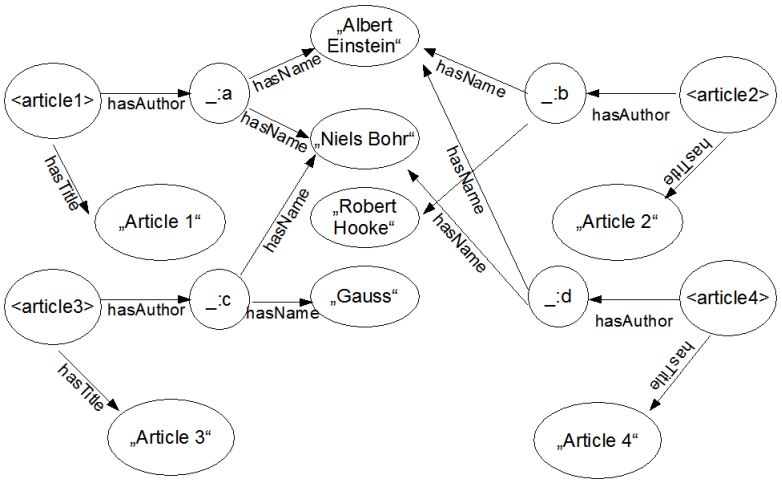



**Fig. 2.** RDF data graph to process the SparQL query provided in Listing 1.

Listing 1 and Figure 2 give examples. Matching the query from Listing 3 against the RDF dataset in Figure 2 returns the subgraphs shown in Figure 3. The final result set, is a projection of the variable *?title* of these subgraphs, i.e., “Article 1”, “Article 2 ”and “Article 4”.

```
?article <hasTitle> ?title.
?article <hasAuthor> ?author.
?author <hasName> "Albert Einstein".
```

**Listing 3.** SparQL query patterns of query in Listing 1

[1] Internationalized Resource Identifier

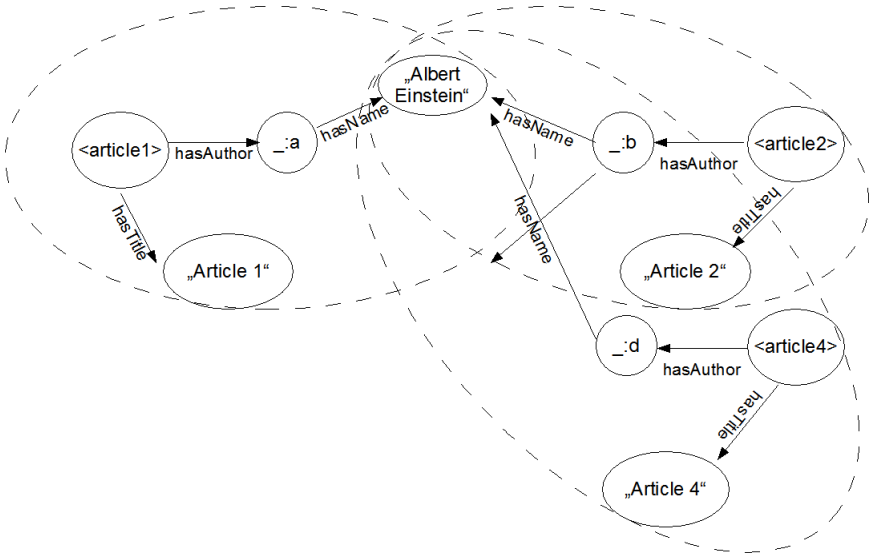



**Fig. 3.** Matching subgraphs of query patterns in Listing 3 over data graph in Figure 2.

# 4 The RDFMatView Approach

We propose the evaluation of SPARQL queries using other queries that were materialized offline. We call those queries materialized queries. In this chapter, we formally introduce all necessary concepts for this idea. Section 4.1 defines materialized queries as indexes and shows how one can decide which of the existing indexes is suitable for a given query. Those indexes are called eligible, and a set of eligible indexes can be combined to cover a query completely or partly. Section 4.2 describes the algorithm which produces all possible covers for a query given a set of indexes. An extensive example to better explain our ideas is presented in Section 4.3.

## 4.1 Patterns, Mappings, Occurrences, Indices and Covers

Before we can define an index over RDF graphs we need to explain the concepts of query pattern, mapping and occurrence of a pattern. Definitions 1, 2 and 3 introduce these concepts respectively.

**Definition 1 (Query Pattern).** *Let $Q$ be a simple SPARQL query. Then, $P(Q)$ denotes its* query pattern*, which is the set of triple patterns in the body of $Q$.*

**Definition 2 (Mapping, total Mapping).** *Let $P$ be a query pattern and $V_P$ the set of variables in $P$. A mapping is a function defined as follows:*

$$S : V_P \rightarrow \mathit{RDF\text{-}T} \cup V$$

*If $S(v) \notin V$ for all $v \in V_P$, then $S$ is a* total mapping.

Our notions of mappings and total mappings is based on the SPARQL-Standard [2] and its definition of *pattern solutions*. However, while in the SPARQL standard such solutions are only searched in the data graph, we also need to permit that variables are mapped to other variables. This generalization allows us to search occurrences of patterns in other patterns, in particular, occurrences of indexes in a query.

**Definition 3 (Occurrences of a pattern).** *Let $P_1$ and $P_2$ be two query patterns. $P_1$ occurs in $P_2$, denoted by $P_1 \sqsubseteq P_2$, iff there is a mapping $S$ such that $S(P_1) \subseteq P_2$. Such $S$ are called* embeddings *of $P_1$ in $P_2$.*

When we speak about a concrete occurrence of an index pattern in a query pattern, we will refer it to as an *embedding* (to contrast from the term occurrences, which we from now on only use for matches of a pattern in the data graph). Figure 4 illustrates an embedding of a pattern $P_1$ in a pattern $P_2$.

Using the previously introduced concepts, we can now define an index over an RDF data graph.

**Definition 4 (Indexes).** *An* index *$I$ over a RDF data graph $G$ is a pair $I = (P, O)$, where $P$ represents a query pattern and $O$ represents the set of all occurrences of $P$ in $G$. $P$ is called the index pattern of $I$. For an index $I = (P, O)$, the* size of *$I$, written as $|P|$, is the number of triple patterns in $P$. The frequency of $I$ in $G$, written as $\#_G(P)$, is the number of occurrences of $P$ in $G$.*

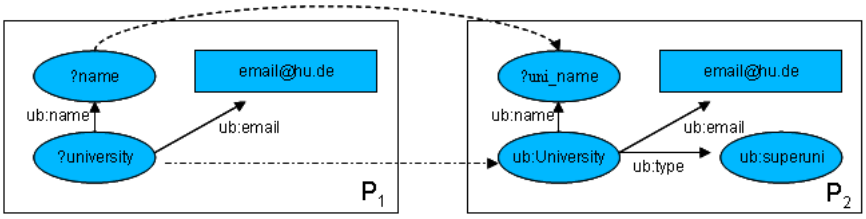



**Fig. 4.** Pattern $P_1$ occurs in $P_2$ using mappings from ?*name* ⇒?*uni_name* and ?*university* ⇒ *ub* : *University*

These notations are used for both indexes and queries, i.e., $|Q| := |P(Q)|$ is the *size of a query* $Q$ and $\#_G(P(Q))$ is its frequency. From here on, the frequency will be denoted as $\#(P)$, if the query and the data graph are clear from the context.

In our approach, indexes are defined offline, i.e., created by an administrator before queries are executed. At query time, the system needs to determine which of the existing indexes are useful for the given query $Q$. Clearly, only those indexes are candidates for speeding up $Q$ whose patterns are contained in $P(Q)$, i.e., indexes which have an embedding in $Q$. We call all such indexes *eligible for* $Q$. The following definition formally captures this idea.

**Definition 5 (Eligible Index).** *Let $G$ be a data graph, $\mathfrak{I}$ the set of indexes on $G$, and $Q$ a query against $G$. We call an index $I \in \mathfrak{I}$ eligible for $Q$ iff $P(I) \sqsubseteq P(Q)$. The set of all eligible indexes for query $Q$ is denoted by $\mathfrak{I}_Q$.*

Definition 5 describes which indexes can be used to help processing a given query. Note that an eligible index can be used in different ways to process a query if it has different embeddings. However, it would be even more advantageous if query processing could use more than one index. The best situation occurs if the patterns of the different indexes overlap (in a sense which will be clear in the following). Overlapping indexes are good candidates for reducing query processing time because the query engine can combine occurrences of these indexes and thus quickly generate solutions for larger fractions of the query pattern.

We define two ways in which indexes can overlap. Two indexes overlap intensionally iff there *could* exist a triple pattern in which their materialization would overlap. In contrast, two indexes overlap extensionally if their materializations overlap on a concrete data graph. Thus, intensional overlap relies only on the pattern of indexes and is independent of a concrete data graph, while extensional overlap needs to consider the actual data graph.

**Definition 6 (Overlapping Indexes).** *Let $I_1 = (P_1, O_1)$ and $I_2 = (P_2, O_2)$ be two indexes over a data graph $G$.*

- *$I_1$ and $I_2$ intensionally overlap iff there exists mapping functions $S_1$, $S_2$ such that*

$$S_1(P_1) \cap S_2(P_2) \neq \emptyset$$

– *$I_1$ and $I_2$ extensionally overlap in G iff*

$$\exists o_1 \in O_1, o_2 \in O_2 : o_1 = o_2$$

However, when we want to use overlapping indexes for processing of a query $Q$, we need to refine our definitions as the query strongly restricts the mappings we need to consider.

**Definition 7 (Overlapping Embeddings).** *Let $I_1 = (P_1, O_1)$ and $I_2 = (P_2, O_2)$ be two indexes over a data graph G. Let Q be a query over G with $P_1 \subseteq Q$ and $P_2 \subseteq Q$, and let $m_1$ be an embedding of $P_1$ in Q and $m_2$ an embedding of $P_2$ in Q.*

– *$m_1$ and $m_2$ intentionally overlap in Q iff*

$$m_1(P_1) \cap m_2(P_2) \neq \emptyset$$

– *$m_1$ and $m_2$ extensionally overlap in Q and G iff*

$$m_1(O_1) \cap m_2(O_2) \neq \emptyset$$

[2]

Computing intensional overlaps can be implemented efficiently as this property is independent from the actual data graph (and updates to it). In contrast, computing extensional overlaps is costly as it requires execution of index queries and comparison of their results on a given data graph. On the other hand, at query execution time information about extensional overlaps would be more important than those about intensional overlaps, as the latter is only a necessary yet not sufficient condition for the existence of a concrete overlap given the query. Actually, if two embeddings intensionally overlap in the query but do not extensionally overlap in the data graph, one can immediately conclude that the query has no answer. However, for the rest of this work we only consider intensional overlaps to avoid the costly pre-computation and maintenance of extensional overlaps.

Using the notion of overlaps, we can finally define the *cover of a query*.

**Definition 8 (Cover).** *Let Q be a query and $E_Q$ the set of all embeddings of eligible indexes for Q in Q. Let $C \subseteq E_Q$. Build a graph $G_C$ for C as follows: Each embedding in C is represented as a node. Whenever two embeddings from C intensionally overlap in Q, we add an edge to $G_C$ between the nodes representing the embeddings. Any C for which $G_C$ has only one connected component is called a* cover for *Q.*

We focus on covers with overlapping embeddings since they allow better estimations of the cost savings that can be achieved with them (see next Section). Furthermore, we are only interested in maximal covers, i.e., those covers which cannot be extended further by adding new embeddings.

[2] With slight abuse of notation. By $m_1(O_1)$ we mean the projection of all occurrences in $O_1$ using $m_1$.

**Definition 9 (Maximal Covers).** *Let $Q$ be a query and $C_1$ , $C_2$ be two covers for $Q$. $C_1$ is subsumed by $C_2$ if $C_1 \subseteq C_2$. Any cover which is not subsumed by another cover is called* maximal.

In the following, we only consider maximal covers. We classify those into two different groups:

- A cover is complete if it covers *all patterns* of a query.
- A cover is partial if it is not complete.

Usually it is not possible to find a complete cover of a query. According to this, we refer in the rest of this article to a partial cover as a cover.

### 4.2 Finding covers

Definition 8 is purely conceptual. We now show how we actually compute the set of indexes and how we combine their embeddings to find all covers.

The first task is performed by a adaption of the classical algorithms for query containment of relational queries [17]. Essentially, we find all mappings between any index pattern and the query pattern by enumerating all possible cases. If an mapping exists then we can conclude that an index is eligible for that query and we store mapping as an embedding. Note that, for a given index, there are potentially many different ways to be eligible, i.e., different mappings between index and query patterns and therefore, multiple embeddings. Algorithm 1 illustrates this process.

**Algorithm 1** Pseudo code for SPARQL query containment. The algorithm computes all embeddings of indexes in a given query.

**Given:** Query $Q$, set of indexes $\mathfrak{I}$
**Returns:** Set $E_Q$ of all embeddings

```
1: P(I), P(Q) {index and query patterns}
2: 𝔖 := ∅ {Set of embeddings of P(I) in P(Q)}
3: E_Q := ∅ {Set of all embeddings}
4: for all Index I in 𝔍 do
5:     𝔖 := S|S(P(I)) = P(Q)
6:     if 𝔖 <> ∅ then
7:         E_Q = E_Q ∪ 𝔖
8:     end if
9: end for
```

The core of algorithm 1 is line 5 which computes all embeddings of an index in the query pattern. The implementation of this step is shown in in Algorithm 2 which traverses a tree representing the search space of all possible mappings from the index into the query. Each level in the tree contains all mappings for a specific triple pattern. All mappings of a level in the tree are children of each mapping of the previous level. In Line 12 we generate this tree and traverse it using backtracking in Line 13. During the traversal, the mappings for the different triples are combined (if compatible) to

increasingly larger mappings. Whenever all triples of the index have been mapped to the query pattern by one mapping, this mapping is added to the set of embeddings.

**Algorithm 2** Pseudo code for searching all embeddings of an index in a query patterns. It maps variables from the index to the query

**Given:** Query pattern $P(Q)$, index pattern $P(I)$
**Returns:** $\mathfrak{O}$, set of embeddings of $P(I)$ in $P(Q)$

```
 1: L_t := ∅ {Temporal list of occurrences of each index triple pattern in P(Q)}
 2: L := ∅ {Total list of t_i occurrences in P(Q)}
 3: for all Triple Pattern t_i in P(I) do
 4:    for all Triple Pattern t_q in P(Q) do
 5:       if t_i occurs in t_q with mapping S then
 6:          L_t := L_t ∪ S
 7:       end if
 8:    end for
 9:    L := L ∪ L_t
10:    L_t := ∅
11: end for
12: occTree := createTree(L)
13: return 𝔒 := traverseTree(occTree)
```

Once we have all embeddings, we proceed to generate covers. To this end, we first compute all mutual intensional overlaps between indexes and store them in a matrix. We then incrementally build maximal covers by finding all maximal connected components in this matrix.

### 4.3 Example

We illustrate all previously introduced concepts using a comprehensive example. Consider the RDF data listed in Figure 5, the SPARQL query $Q_1$ shown in Listing 4, and the two RDFMatView indexes $I_1, I_2$ from Listing 5 and 6, respectively.

```
SELECT * WHERE {
    ?university  rdf:type ub:University;
       ub:name ?university_name.
    ?ub_department rdf:type ub:Department;
       ub:name ?ub_name_department;
       ub:subOrganizationOf ?university .
}
```

**Listing 4.** SPARQL query $Q_1$ computing universities and their departments

```
SELECT * WHERE {
     ?place  rdf:type ?place_type;
          ub:name ?place_name.
}
```

**Listing 5.** RDFMatview $I_1$ computing places and their names

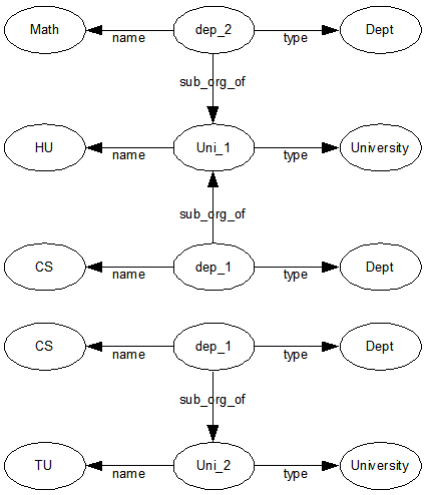



**Fig. 5.** RDF data set

```
SELECT * WHERE {
 ?ub_department rdf:type ub:Department;
      ub:name ?ub_name_department;
      ub:subOrganizationOf ?university;
 ?university  rdf:type ub:University.
}
```

**Listing 6.** RDFMatview $I_2$ computing universities with their departments

Executing $Q_1$ on the data set in Figure 5 produces the result shown in Table 1.

**Table 1.** Result of $Q_1$.

| **Query Result Set** | | | |
|---|---|---|---|
| **university** | **university_name** | **ub_department** | **ub_name_department** |
| Uni_1 | HU | dep_1 | CS |
| Uni_1 | HU | dep_2 | Math |
| Uni_2 | TU | dep_1 | CS |

Materializing $I_1$ and $I_2$ produces results as given in Table 2 and Table 3, respectively.

Both $I_1$ and $I_2$ are eligible for the query. Actually, $I_1$ is eligible in two different ways, as it may either substitute the link between a university and its name or the link between a department and its name. The three resulting embeddings are shown in Table

**Table 2.** Result of $I_1$.

| **Index1** | | |
|---|---|---|
| **place** | **place_type** | **place_name** |
| Uni_1 | University | HU |
| Uni_2 | University | TU |
| dep_1 | Dept | CS |
| dep_2 | Dept | Math |
| dep_1 | Dept | CS |

**Table 3.** Result of $I_2$.

| **Index2** | | |
|---|---|---|
| **ub_department** | **university** | **ub_name_department** |
| dep_1 | Uni_1 | CS |
| dep_2 | Uni_1 | Math |
| dep_1 | Uni_2 | CS |

4 and Table 5.

**Table 4.** Embeddings of $I_1$ in $Q_1$.

| **Index** | | **Query** |
|---|---|---|
| **Embedding 1** | | |
| ?place | ⇒ | ?university |
| ?type | ⇒ | ub:University |
| ?place_name | ⇒ | ?university_name |
| **Embedding 2** | | |
| ?place | ⇒ | ?ub_department |
| ?type | ⇒ | ub:Department |
| ?place_name | ⇒ | ?ub_name_department |

The first embedding of index $I_1$ intensionally overlap with the embedding of index *I*2 in the triple pattern *?university rdf:type ub:University*. Thus, these two embeddings form a cover (in this case the only cover with more than one embedding).

Assume for now the RDF data would be stored in a RDBMS within a single *triple* table, for instance, *Triple(subj, prop, obj)*. Hence, the query in Listing 4 could be answered by the SQL query described in Listing 7.

```
SELECT t1.subj AS a0, t2.obj AS a1, t3.subj AS a2, t4.obj AS a3
```

**Table 5.** Embeddings of $I_2$ in $Q_1$.

| Index | | Query |
|---|---|---|
| ?ub_department | ⇒ | ?ub_department |
| ?ub_name_department | ⇒ | ?ub_name_department |
| ?university | ⇒ | ?university |

```
FROM Triple AS t1, Triple AS t2, Triple AS t3,
     Triple AS t4, Triple AS t5
WHERE t1.prop='type' AND t1.obj='University' AND
      t2.prop='name' AND t3.prop='type' AND
      t3.obj='Department' AND t4.prop='name' AND
      t5.prop='subOrganizationOf' AND
      t1.subj = t2.subj AND t3.subj = t4.subj AND
      t3.subj = t5.subj AND t1.subj = t5.obj;
```

**Listing 7.** SQL to answer the query from Listing 4

Listing 7 shows that four self joins are required. However, using the pre-computed data for Index1 and Index2, one can answer the query with only one join, as shown in Listing 8.

```
SELECT index2.university,
       index1.place_name AS university_name,
       index2.ub_department,
       index2.ub_name_department
FROM index1, index2 WHERE
    index1.place = index2.university;
```

**Listing 8.** SQL representation of SPARQL query in Listing 4 using RDFMatView indexes

This example illustrates that it may make sense to use materialized queries as indexes to process SPARQL queries. Note that in this special example we can actually answer the query only from the materialized query and thus do not need to access the RDF database at all, because we query pattern can be completely covered. However, in of complete cover exists, the results of a cover must be combined appropriately with queries against the RDF database that compute results of those parts of the query that are left uncovered. Section 6 will describe this process in detail.

## 5 Cost Model

In the previous sections we defined which indexes and which sets of indexes, i.e., covers, are eligible for a given query. At run time, the optimizer must choose between these different options, or decide to execute the query without using indexes. This decision should be taken based on the expected savings in time that the usage of one or more indexes brings to query execution. In the following, we define a simple model for estimating these savings. This model implicitly makes a number of assumptions on the data graph. For instance, we treat all triples of a pattern equally with respect to their expected numbers of matches, independent of whether or not the triple contains variables, and independent of the real frequency of constants. These assumptions make the model very simple and also allow us to estimate query costs without any detailed knowledge of the underlying database. Clearly, finding more detailed cost models is an important future work.

Our model estimates the cost of executing a query with zero, one, or more indexes. Note that it is not the purpose of the model to directly estimate the necessary execution time, as it depends on a multitude of factors which are extremely difficult to model (such as processing strategy, size of data sets, hardware etc.). In contrast, our model only aims at discerning good plans from bad plans; to this end, the estimated cost only must correlate with the real time. We shall evaluate the quality of our cost model empirically in Section 7.

Our model is based on the following fundamental observations. For each index $I$ that occurs in the query pattern, each occurrence of the query pattern in the data graph must contain an occurrence of $I$. This leads to following facts:

1. It makes sense to prefer those indexes which have few occurrences because every occurrence of an index must be validated to verify the possibility to extend it to an occurrence of the query.
2. It is reasonable to cover as much as possible from the query patterns. This process reduces the number of query patterns that need to be evaluated against the data graph. According to this, large index patterns are specially interesting.

We formally capture these observations in the definition of the *selectivity* of an index. It defines the relation of the number of occurrences of an index in a given graph to the possible total number of index occurrences in the graph. To calculate selectivity, we need the size and the frequency of the index pattern as well as the size of the data graph.

**Definition 10 (Selectivity of an index).** *Let $I$ be an index over a data graph $G$. The selectivity $s(I)$ of $I$ is defined as:*

$$s(I) = \frac{\#(I)}{|G|^{|I|}}.$$

We derive our formula for estimating the selectivity of a set $I$ of indexes from the previous definition. To this end, we view $I$ as the union of the patterns of the indexes in $I$ (similar to the union of RDF graphs, see [18]). Without further knowledge, the

selectivity of $I$ is worse than the selectivity of all its indexes, because any occurrence of one index potentially can be combined with any occurrence of all other indexes. The leads to the following worst-case estimation for the selectivity of a set of indexes.

**Lemma 1 (Selectivity of a set of indexes).** *Let $G$ be an RDF data graph and $I = \{I_1, ..., I_n\}$ with $I_i = (P_i, O_i), i = 1, ..., n$ be a set of indexes over $G$. We define the selectivity of $I$ as:*

$$sel(I) = sel(I_1 \cap I_2 \cap \ldots \cap I_n) = \frac{\prod_{i=1}^{n} |O_i|}{|G|^{\max\{|P_1|,...,|P_n|\}}}$$

*Proof.* As any occurrence of one index in the worst case is combined with any occurrence of any other index, it follows that $sel(I) \leq sel(I_1) \cdots sel(I_n)$. Further, the size of the index pattern of $I$ is at least $|P_1 \sqcup ... \sqcup P_n| \geq \max\{|P_1|, ..., |P_n|\}$. Together, we have:

$$sel(I_1 \sqcup I_2 \sqcup \cdots \sqcup I_n) \leq \frac{\prod_{j=1}^{n} |O_i|}{|G|^{\{|P_1 \sqcup ... \sqcup P_n|\}}} \leq \frac{\prod_{j=1}^{n} |O_i|}{|G|^{\max\{|P_1|,...,|P_n|\}}}$$

□

Lemma 1 assumes that we do not have any information about relationships between indexes of this set. However, we already defined ways in which indexes may overlap (see Definition 6); furthermore, for query processing we restricted ourselves to covers, i.e., sets of overlapping embeddings (see Definition 7). Knowledge about overlaps between embeddings allows for a more accurate estimation of the selectivity of a cover.

As explained in Section 4 intensional overlapping is only a necessary yet not sufficient condition for the existence of a concrete overlap on the underlying query. Actually, if two embeddings intensionally overlap in the query but do not extensionally overlap in the data graph, one can immediately conclude that the query has no answer. Therefore, there are two different cases which should be considered for the selectivity of a cover: i) The embeddings overlap intensionally but not necessarily extensionally and ii) the embeddings overlap intensionally and extensionally.

If embeddings overlap intensionally but not necessarily extensionally, selectivity can be estimated similarly to selectivity of a set of indexes (see Lemma 1), since an embedding can be seen as an instance of its underlying index. Thus, any occurrence of one embedding in the worst case can be potentially combined with any occurrence of any other embedding.

**Lemma 2 (Selectivity of intensionally overlapping embeddings).** *Let $G$ be an RDF data graph, $Q$ a query over $G$ and $m = \{m_1, ..., m_n\}$ a cover of $Q$. Then, we can estimate the selectivity of $m$ as follows:*

$$sel(m) = sel(m_1 \cap m_2 \cap \ldots \cap m_n) = \frac{\prod_{i=1}^{n} |m(O_i)|}{|G|^{\max\{|m_1|,...,|m_n|\}}}$$

For the second case, when the cover consists of intensionally overlapping embeddings that also overlap extensionally, we can sometimes use a stronger estimation:

**Lemma 3 (Selectivity of extensionally overlapping embeddings).** *Let G be a data graph , Q a query over G and $m = \{m_1, ..., m_n\}$ a cover of Q. Assume that all pairs of embeddings also mutually overlap extensionally. Then, we can estimate the selectivity of m as follows:*

$$sel(m_1, ..., m_n) \leq \frac{min(|m(O_1)|, ..., |m(O_n)|)}{|G|^{\max\{|m_1|,...,|m_n|\}}}$$

*Proof.* Because all pairs of embeddings overlap extensionally (see Definition 7), at most

$$min(|m(O_1)|, ..., |m(O_n)|)$$

occurrences are selected.

However, as state previously, for this paper we only consider intensionally overlapping embeddings. Thus, in the following we estimate the selectivity of a cover using Lemma 2.

Having computed the selectivity of all maximal covers, the query optimizer must determine which cover is the best for query processing. Assessing the cost of a cover is not enough for this purpose, as we need to estimate the cost of a query given a cover. We distinguish two cases: i) The cover completely covers the query, or ii) the cover only partially covers the query. The first case is clear. The cost of a query given a complete cover is the same as the estimated cost of the cover. The second case is more interesting since the residual part of the query muss be taken into account. Note that this part depends on the query, and therefore no offline estimations are possible.

We propose a cost model for this case which treats the residual part of the query as an index. However, since we do not have any information about this part, we propose to estimate the frequency of the pattern using the size of the pattern and the dataset. The idea is to estimate the frequency of a pattern dividing the total number of triples contained in the dataset between the number of triple patterns contained in the residual part of the query. Assume an RDF dataset containing 150K triples and two patterns $P_1$ and $P_2$ containing two and three triples respectively. The estimated frequencies for $P_1$ and $P_2$ are 75,000 and 50,000 respectively, which captures our expectation that a pattern with more triples will have less matches. According to this we define the frequency of a residual part of a query as follows:

**Definition 11 (Frequency of the residual part of a query).** *Let G be an RDF dataset and R a query pattern. The frequency $\#_G(R)$ is defined as the ratio of the size of G and the size of R.*

$$\#_G(R) = \frac{|G|}{|R|}$$

According to Definition 10, the selectivity of an index is given by

$$s(I) = \frac{\#(I)}{|G|^{|I|}}.$$

Applying our definition of frequency of the residual part of the query into the above model we obtain:

$$s(R) = \frac{\frac{|G|}{|R|}}{|G|^{|R|}}.$$

simplifying the equation, results in the following model:

**Definition 12 (Selectivity of the residual part of a query).** *Let $G$ be an RDF dataset and $R$ a query pattern. The selectivity of $R$ is defined as:*

$$s(R) = \frac{1}{|R| \cdot |G|^{|R-1|}}.$$

Finally, based on Lemma 2 and Definition 12 we define the estimated cost for a SPARQL query given a cover as follows.

**Definition 13 (Estimated cost for a query given a cover).** *Let $Q$ be a SPARQL query, $C$ a cover of $Q$ and $R$ the residual patterns of $Q$, i.e., the triples from $P(Q)$ not covered by $C$. Then, we estimate the cost of execution $Q$ using $C$ as*

$$c(Q, C) = sel(C) \cdot sel(R)$$

# 6 Implementation

In this section, we describe how our approach can be integrated into an existing SPARQL query processor. Such an integration touches upon several components of a system: First, we need to be able to execute index queries and to store their results (plus some metadata) persistently. To use indexes in query processing, we need to intercept the query processor to, at the right point in time, search for an optimal cover. Finally, we must change the way how queries are executed, as we need to divide the query pattern into that part that is covered by the chosen cover - which is answered by retrieval of the materialized information - and the rest of the query pattern. We present solutions to all these steps for the ARQ system [19]. However, we want to stress that general process would be the same for any other SPARQL query processors.

In this chapter, we first give some details on ARQ and Jena, its storage model. We then provide a high level description of our approach. Next, we show how an index is made persistent using a data dictionary for saving space. Finally, we describe three ways in which query processing in ARQ can integrate materialized queries as indexes. Those will be evaluated separately in the next chapter.

## 6.1 ARQ and the Jena Persistent Storage Schema

For our integration with ARQ we use the Jena persistence subsystem. This subsystem implements the Jena Model interface using a back-end relational database engine. The default Jena database layout uses a denormalized schema centered around a statement table which essentially stores every RDF tripel as tuple. However, the values in the triple can either be included as value, or they are stored in other tables. Specifically, *short* literals are stored directly in the statement table, while *long* literals are stored in a literal table. Similarly, short URIs are stored in the statement table and long URIs are stored in a resources table. Table 6 and Table 7 describe the layout for those tables. Though this scheme helps to reduce space requirements especially in the presence of long and frequently used URIs or labels, it makes query processing more complicated as, for each row in the statement table, one must decide at runtime whether the respective value can be obtained directly or if a a join to another table is necessary. that it stores reified statements in an optimized form. Recall that a reified statement is expressed in RDF as four individual RDF statements. Storing this would require four rows in the standard representation, while the reified statement table stores each statement in a single row. For applications that use a large number of reified statements, the space savings can be substantial. Additionally, Jena defines system tables to store meta data. For further information we refer the reader to [3].

## 6.2 Implementation overview

We differentiate two phases. At offline-time, indexes are created, analyzed, and their results are materialized. At query-time, queries are answered with the help of indexes. We divide our description of our implementation according to these phases.

**Index creation.** Indexes are created offline. Upon creation of an index, the following things happen. First, a new table is created which will store the materialized query. The

**Table 6.** Jena statement table for asserted (non-reified) statements.

| Column | Type | Description |
|---|---|---|
| Subj | Varchar not null | Subject of asserted statement (ID or value) |
| Prop | Varchar not null | Predicate of asserted statement (ID or value) |
| Obj | Varchar not null | Object of asserted statement (ID or value) |
| GraphId | Integer | Identifier of graph (model) that contains the asserted statement |

**Table 7.** Jena long literals table storing literals that are considered as too long to directly be stored in the statement table.

| Column | Type | Description |
|---|---|---|
| Id | Integer not null | Identifier of long literal, referenced from the statement tables |
| Head | Varchar not null | First n characters of long literal (encoded) |
| ChkSum | Integer | Checksum of tail of long literal |
| Tail | Blob | Remainder of long literal (long literal without the head) |

schema of this table is specific to the index query: Each different variable contained in the index is represented as a field. Next, the query is executed, which leads to bindings for those variables. These are stored in the respective fields (see Section 6.4). At the end, every tuple in that table represents one result to the materialized query. During this process, we also create a data dictionary which relates resource to unique identifiers (see Section 6.3). We only store those IDs in the index tables; this scheme is similar to the one used in Jena (see above), but we omit the costly choice between included and external values. These steps are executed only once per index (recall that index updates are beyond the scope of this work).

**Index usage.** At query-time, queries are analyzed and answered, possibly by using one or more of the materialized indexes. This breaks down into the following steps:

1. Analysis of the query to find all maximal covers
2. Selection of the most suitable cover to answer the query given our cost model
3. Rewriting of the query using the chosen cover
4. Extension of the results of the cover to results of the query

Steps one and two were already discussed in Section 4. Here, we concentrate on the third step, the query rewriting. Query rewriting can be performed in three different ways: i) using only ARQ, ii) by translation into SQL and access to the Jena native storage tables, and iii) by using a combination of ARQ and SQL. These different options will be discussed in Section 6.5.

### 6.3 RDF Data Dictionary

As triples may contain long string literals, we generate a data dictionary that maps all different RDF terms to a unique ID. This has two main benefits: 1) It decreases the space requirements of indexes, and 2) it is a simplification for the query processor since it will have to deal only with numerical values instead of string values. This makes a difference as, depending upon the specific strategy for query processing chosen, values might have to go back-and-forth between the database and the query processor. The cost for this gains is that at the end of query evaluation, all IDs need to be translated into the original values.

Listing 8 illustrates the schema of our RDF data dictionary.

**Table 8.** RDF Data Dictionary

| Column | Type | Description |
|---|---|---|
| IDResource | bigint not null | Resource identifier (encoded) |
| Resource | Varchar not null | Original resource value |

### 6.4 RDFMatView Index processing

Each index is materialized as a proper table in the underlying relational database. Its schema is formed by the set of different variables contained in the index, regardless of whether the variables are contained in the SELECT clause of the query or not. Occurrences of the index in the data set are stored as values for these fields. Each attribute of one tuple represents a binding for the respective variable, which is represented as an ID from the RDF data dictionary. An example is shown in Listing 9.

```
CREATE TABLE Index1 (
    place       bigint REFERENCES Dictionary_RDF (IDResource),
    place_type  bigint REFERENCES Dictionary_RDF (IDResource),
    place_name  bigint REFERENCES Dictionary_RDF (IDResource)
);
```

**Listing 9.** Materialization of the index from Listing 5 as RDFMatView index

During the processing of a SPARQL index we also calculate and store index properties, for instance size and frequency, which are used later to assess query execution plans. This information is stored in a single metadata table.

### 6.5 Executing a query using RDFMatView indexes

Query processing using *RDFMatViews indexes* usually combines results of multiple indexes. However, it is not always possible to cover all patterns of the query. The set of uncovered patterns is referred to as *residual part of a query*. To completely answer a query, it is necessary to extend the results of the selected indexes with the results for the residual part of the query. We developed three different strategies to fulfill this task:

– Our first strategy uses ARQ to process the residual part of the query. RDFMatView extends the results of the chosen cover by joining the partial solutions with the solutions of the residual patterns.
– The second strategy is based on a SPARQL-to-SQL algorithm. The idea is to directly access the native Jena storage tables and to combine those results with the index tables to generate the final result set.
– The last strategy is a combination of the previous two strategies, i.e. ARQ query engine and database execution engine.

These strategies are explained in detail in the next sections.

### Method 1: MatView-and-ARQ Engine

MatView-and-ARQ is a rewriting engine built on top of the Jena Framework. Given a query and a cover, it computes the set of residual patterns of the query and uses ARQ to execute this (sub-)query. Furthermore, it computes the result of the cover by joining the respective index tables according to the variable mappings of the embeddings forming the cover. Results are also joined with the data dictionary to obtain RDF values, and finally joined to the result of the ARQ query to produce the complete answer to the original query. This engine encapsulates the logic for the execution of the cover and provides total independence from the underlying relational database system. Figure 6 illustrates the workflow of this engine.

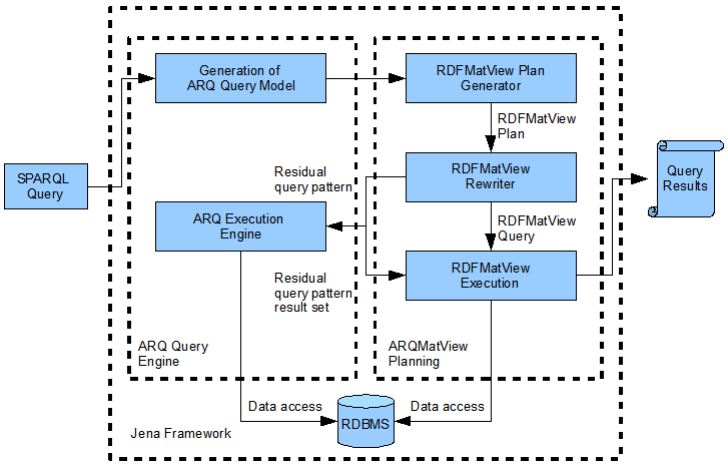



**Fig. 6.** Workflow of the query processing using Matview-and-ARQ.

### Method 2: MatView-to-SQL Engine

MatView-to-SQL is a rewriting engine which, unlike our first method, translates the residual part of the query into a SQL query on the Jena tables using an algorithm proposed by Chebotko in [20]. The SQL query is executed by the RDBMS. The result set is processed using our RDF Dictionary and finally combined with the results of the cover. The complete query processing is performed inside the database execution engine using a stored procedure. Figure 7 illustrates the workflow of this engine.

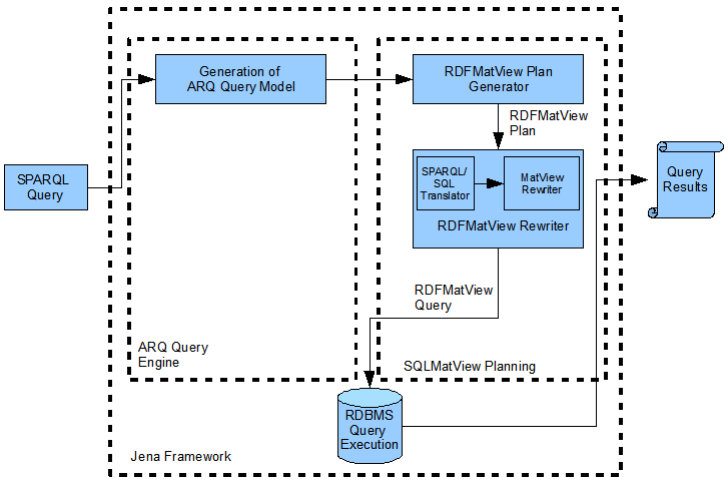



**Fig. 7.** Workflow of the query processing using MatView-to-SQL

### Method 3: Hybrid Engine

The third method is a mixture of MatView-and-ARQ and MatView-to-SQL. As in Method 1, after rewriting the query, this engine transfers the residual patterns to the query execution engine of ARQ. The second part of the process combines the results of the residual patterns with the resulting set of the covered part of the query patterns. However, contrary to Method 1, this engine is database-dependent since this task is performed inside the database execution engine, as in Method 2. Figure 8 illustrates the workflow of this engine.

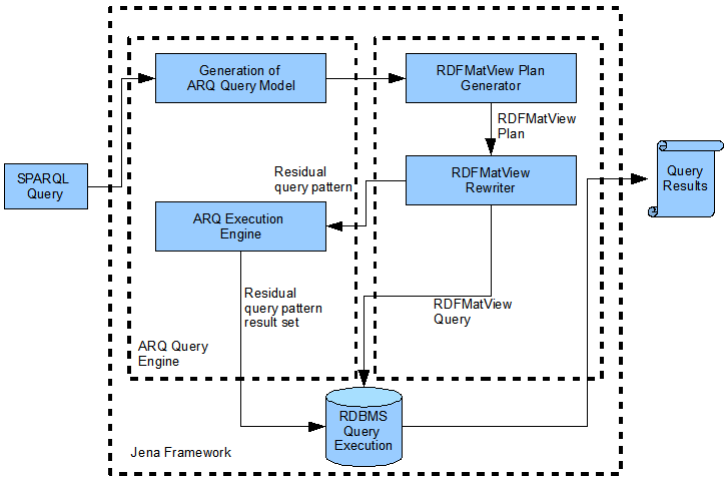



**Fig. 8.** Workflow of the query processing using our hybrid engine.

# 7 Evaluation

In this section, we describe a preliminary evaluation of our approach using the Berlin SPARQL Benchmark. This benchmark allows the creation of data sets with configurable sizes [21]. We generated five RDF datasets with sizes ranging from 250K to 25M triples and tested the respective impact of indexes using three queries from the benchmark set. For each of these queries, we created a range of different indexes, leading to covers composed of one to three embeddings. Note that it is not our intention to find the best set of indexes given a workload (which is generally called index selection, see, e.g., [22–24]); instead, we want to study to which degree different indexes using different processing schemes speed up the execution of queries. As SPARQL processor, we use the ARQ/Jena RDF Storage System (version 2.5.7) on Postgres 8.2.

In the following, we first briefly introduce the Berlin SPARQL Benchmark. We then describe the datasets and test queries as well as the indexes we used. Finally, we present and discuss the results of our evaluation.

## 7.1 Berlin SPARQL Benchmark

The Berlin SPARQL benchmark is built on an e-commerce use case in which a set of products is offered by different vendors and where consumers have posted reviews about products [21]. The main classes of its schema are:

- *Product* Captures products with different sets of properties and features.
- *ProductType* Classifies products into a hierarchy.
- *ProductFeature* Represents product features for a specific product depending on the product type. Each product type in the hierarchy has a set of associated product features, which leads to some features being very generic and others being more specific.
- *Producer* Represents the producer of products.
- *Vendor* Represents the supplier of products.
- *Offer* Describes an offer to a product.
- *Person* Captures all person-related information.
- *Review* Provides ratings of a product.

The benchmark provides a data generator which supports the creation of arbitrarily large datasets using the number of products as scale factor. Table 9 provides a detailed description of the datasets using three different scale factors, and Figure 9 illustrates a triple representation of the generated data.

The benchmark also defines a set of SPARQL queries to simulate a use-case driven workload. This set emulates the search and navigation pattern of a consumer looking for a product. Basically, the sequence of queries performs the following operations:

1. A consumer searches for products that have a set of general features.
2. From the returning set of products, the consumer has a better idea of what he wants and restricts his search with more specific features.
3. The consumer starts to look at specific products and recent reviews for these.

4. To check the trustworthiness of the reviews, the consumer retrieves background information about the reviewers.
5. The consumer decides which product to buy and starts to search for the best price for this product offered by a vendor that is located in his country and is able to deliver within three days.
6. After choosing a specific offer, the consumer retrieves all information about the offer and then transforms the information into another schema in order to save it locally for future reference.

These use cases are reflected by a total of 12 different queries.

**Table 9.** Berlin SPARQL benchmark. Scaling and dataset population. The number of products is used as scale factor.

| **Number of Products** | **666** | **2,785** | **70,812** |
|---|---|---|---|
| **Number of RDF Triples** | 250,000 | 1,000,000 | 25,000,000 |
| **Number of Producers** | 14 | 60 | 1,422 |
| **Number of Product Features** | 2,860 | 4,745 | 23,833 |
| **Number of Product Types** | 55 | 151 | 731 |
| **Number of Vendors** | 8 | 34 | 722 |
| **Number of Offers** | 13,320 | 55,700 | 1,416,240 |
| **Number of Reviewers** | 339 | 1,432 | 36,249 |
| **Number of Reviews** | 6,660 | 27,850 | 708,120 |
| **Number of Instances** | 23,922 | 92,757 | 2,258,129 |
| **File size Turtle (unzipped)** | 22 Mb | 86 Mb | 2.1 Gb |

### 7.2 Dataset and queries

The performance of our solution is evaluated over five data sets containing 250K, 500K, 1M, 10M and 25M triples. As these data sets have identical value distributions but different sizes, evaluation can fully concentrate on the scalability of our methods. Based on their number of triple patterns, we chose three of the queries from the benchmarks set for experimentation. We transformed the query patterns into simple graph patterns (the only form of patterns our current implementation can cope with) and removed bindings to variables. Bounded variables incur extremely high selectivity resulting in the retrieval of only a handful of triples. Such queries are well supported by existing index structures in RDBMS and do not require the type of join-optimization that is achieved with our optimization technique; therefore, performance gains would be only marginal.

Our test queries are the following (see Appendix A.1 for full details):

- Query1: Finds a complete list of products with a set of generic features.
- Query2: Retrieve all basic information related to the list of products.
- Query3: Retrieve in-depth information about products including offers and reviews.

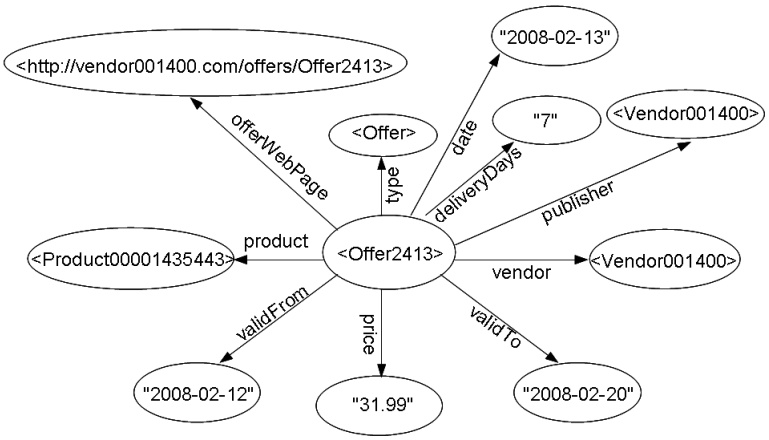



**Fig. 9.** Triple representation of the data set. Publisher and publication data are captured for each instance by a *publisher* and a *date* triple.

The query patterns contain 5, 12 and 13 triple patterns and 6, 12 and 12 different variables, respectively. Afterwards, based on the patterns of these queries, we generate a set of 10 indexes. Indexes are constructed such that they often lead to covers containing a combination of indexes that do not cover the query entirely, since most real-life SPARQL queries fit this case. Indexes were derived using the following rules:

1. Indexes must be completely subsumed by at least one test queries.
2. None of the indexes should completely cover any of the test queries (this case would be trivial).
3. Indexes should have different embeddings that should be intensionally overlapping.

The size of the index patterns varies from 2 to 9 triples. As an example, indexes in Table 10 were derived from Query1. Appendix A.2 gives a complete description of all indexes.

These indexes are designed not only to cover a part of the query, but also to generate multiple embeddings in the query pattern by means of different mappings. The generation of such embeddings allows us to measure the performance of the query processing when using the same participating indexes multiple times covering different parts of the query pattern.

### 7.3 Experiments

In this section, we report the results of our experimental study with the data sets, queries and indexes described in Section 7.2. We evaluated three queries on five different data sets using our three RDFMatView methods and plain ARQ (without indexes), which amounts to 45 different configurations. In this section, we refer to the different approaches to query execution as M1 for MatView-and-ARQ, M2 for MatView-to-SQL,

**Table 10.** RDFMatView indexes derived from Query1.

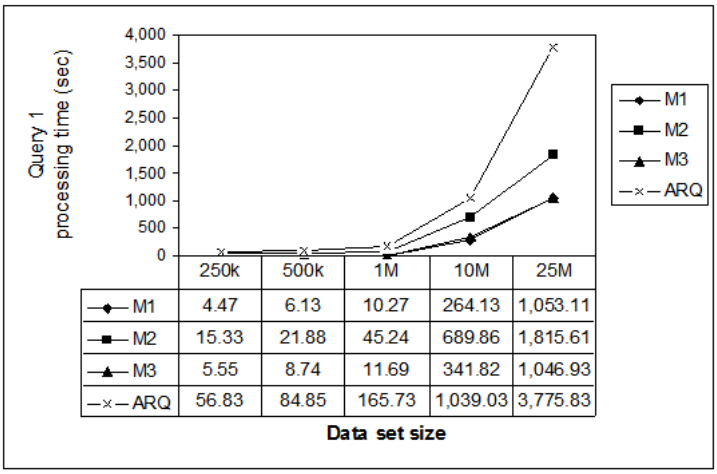
```
Index1: SELECT  * WHERE {
            ?product rdfs:label ?label ;
            rdf:type ?ProductType ;
            bsbm:productFeature ?ProductFeature1. }
Index2: SELECT * WHERE {
            ?product rdf:type ?ProductType;
            bsbm:productPropertyNumeric1 ?value1 .}
```

M3 for the hybrid approach, and ARQ for plain ARQ. We performed experiments using the optimal cover and also evaluated the real and estimated costs of different covers for the same query. Optimal covers were selected according to Lemma 2 (see Section 5). All queries were executed 5 times and average execution times are reported.

Figure 10 illustrates the average processing times for each of the 45 configurations. Clearly, processing time significantly improves when using MatView-and-ARQ (M1) and Hybrid (M3). The improvements are the higher, the larger the database. Processing time does not improve significantly when using MatView-to-SQL (M2). The reason for this is the Jena native storage schema (see Section 6.1). Since the values are encoded following the Jena layout, our process needs to parse the stored values and extract the required information. This process must be performed for each value associated to an exported variable of the query.

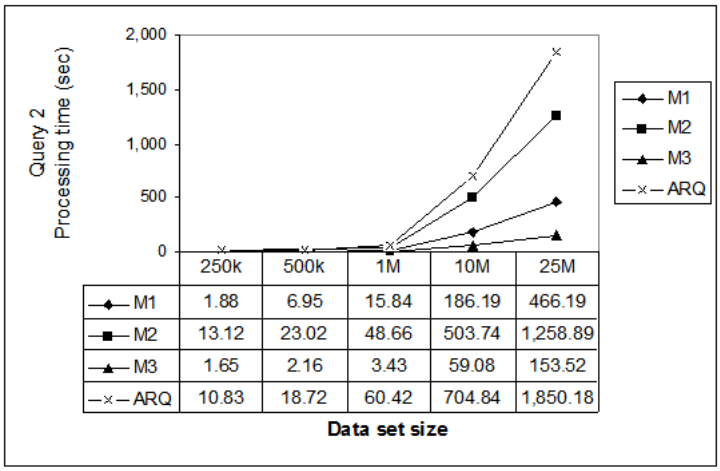
A comparison of real and estimated costs for different covers for Query1 and Query2 are shown in Figure 11 and Figure 13, respectively. Additionally, we analyze in Figure 12 and Figure 14 the relation between the estimated costs of a cover, the number of embeddings it contains and the number of covered and uncovered query patterns. In these figures, covers are sorted descending order first by number of covered patterns, second by number of participating embeddings and third by number of residual patterns. This ordering allow us to verify the correlation between the estimated costs and the number of covered patterns. It also evidences the influence of the number of participating embeddings and residual patterns in the query processing time. Note that Cover 6 in Figure 11 (for Query1) and cover 3 in Figure 13 (for Query2) are those that our system selects as optimal.

Figure 11 and Figure 13 show that the costs estimated by our cost model roughly correlate with the real processing time. However, they also show that there is ample room for improvements. For instance, our model does not yet reflect that using less indexes is advantageous as this requires less joins at runtime; this fact is captured only indirectly by our model as we concentrate on the number of covered patterns. Nevertheless, the results give evidence that our model helps in avoiding the usage of bad plans. Actually all plans improve the total execution times when compared to those without using indexes.



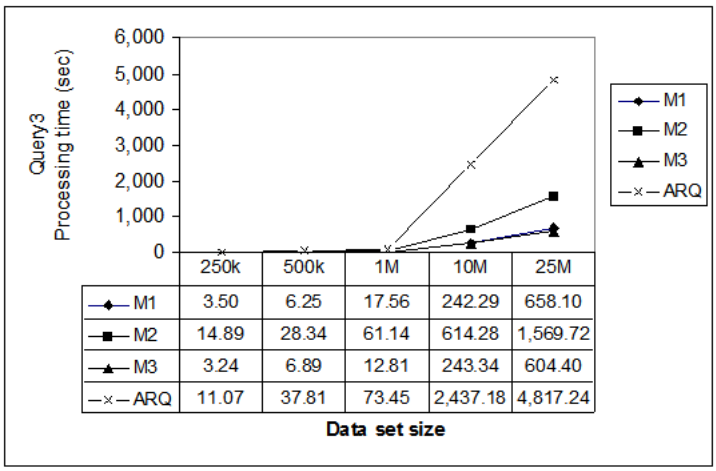

| | 250k | 500k | 1M | 10M | 25M |
|---|---|---|---|---|---|
| M1 | 4.47 | 6.13 | 10.27 | 264.13 | 1,053.11 |
| M2 | 15.33 | 21.88 | 45.24 | 689.86 | 1,815.61 |
| M3 | 5.55 | 8.74 | 11.69 | 341.82 | 1,046.93 |
| ARQ | 56.83 | 84.85 | 165.73 | 1,039.03 | 3,775.83 |

(a) Processing times for Query1




| | 250k | 500k | 1M | 10M | 25M |
|---|---|---|---|---|---|
| M1 | 1.88 | 6.95 | 15.84 | 186.19 | 466.19 |
| M2 | 13.12 | 23.02 | 48.66 | 503.74 | 1,258.89 |
| M3 | 1.65 | 2.16 | 3.43 | 59.08 | 153.52 |
| ARQ | 10.83 | 18.72 | 60.42 | 704.84 | 1,850.18 |

(b) Processing times for Query2




| | 250k | 500k | 1M | 10M | 25M |
|---|---|---|---|---|---|
| M1 | 3.50 | 6.25 | 17.56 | 242.29 | 658.10 |
| M2 | 14.89 | 28.34 | 61.14 | 614.28 | 1,569.72 |
| M3 | 3.24 | 6.89 | 12.81 | 243.34 | 604.40 |
| ARQ | 11.07 | 37.81 | 73.45 | 2,437.18 | 4,817.24 |

(c) Processing times for Query3

**Fig. 10.** Processing time for Query1, Query2 and Query3 on five data sets using three rewriting methods. M1: MatView-and-ARQ; M2: MatView-to-SQL; M3: Hybrid; ARQ: plain ARQ.

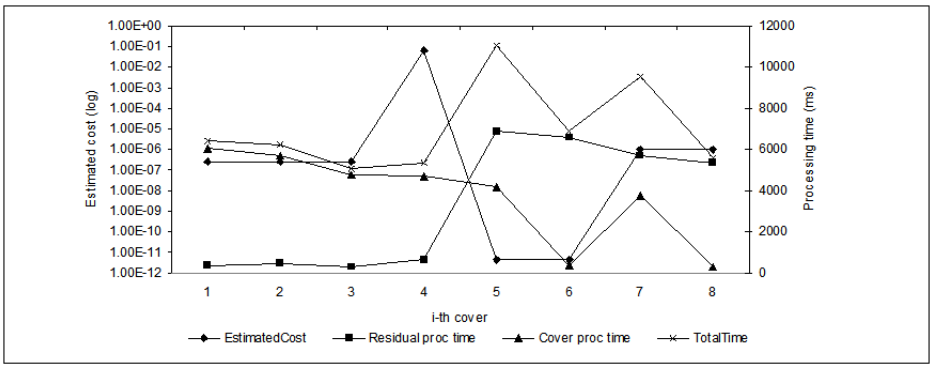



**Fig. 11.** For Query1: Estimated cost, total real processing time, real processing time for retrieving the materialized results, and real procesing time for computing the residual of the query. Values are plotted on log-scale. Note that total real processing time virtually equals real processing time for the covers for larger covers.

A more detailed evaluation can be performed analyzing Figure 12 and Figure 14. Figure 12 illustrates that plans with fewer embeddings covering larger number of patterns have a superior performance. Evidently, having fewer embeddings leads to less time required to join partial results. At the same time, covers with small numbers of occurrences require less time to extend the partial results to results of the complete given query (not visible in the figures).

In Figure 12, covers using 1 or 2 embeddings cover 2, 3 or 4 triple patterns. Covers 5 and 6 have the best estimated costs according to our model. This is reasonable since these covers contain only one embedding and cover 3 out of 5 query patterns. However, the residual part of the query (2 triple patterns) incur an undesirable overhead, which is not yet properly reflected in our model (compare the runtimes in Figure 11). An interesting fact can be observed for those covers covering larger patterns using two embeddings (see covers 1, 2 and 3). These cases show the reduction of processing time when joining two embeddings. At the end, more patterns are covered and the number of patterns to match against the data set decreases. Though their cost estimation is not the best, their processing times are significantly better than those of covers with an estimated better cost. We attribute this behavior to the join (between embeddings) and the processing of the residual part of the query which decreases the fewer are the patterns.

Similar conclusions can be drawn by analyzing the results shown in Figure 13 and Figure 14 for Query2.

As for Query1, the estimated costs and the real processing time approximately correlate as seen in Figure 13. Additionally, the graphic shows that the residual part of the query should be considered as a deciding factor when selecting an optimal cover, since residual processing time nearly spans the complete total processing time (notable especially with larger number of residual patterns. In the graphic, residual patterns range from 3 to 10 patterns.). Figure 14 supports this conclusion showing the details for the

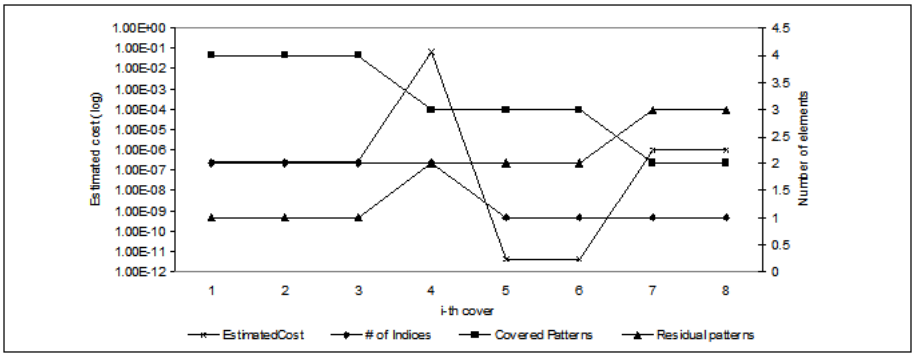



**Fig. 12.** Estimated costs for each generated cover based on intensional dependency relation between indexes for Query1. Costs are based on the cost model introduced in Section 5 and are presented in relation to the size for each cover, number of participating embeddings and the size of the residual part of the query(number of uncovered query patterns). This analysis shows the influence of these elements in the selection of an optimal cover.

generated covers. Such covers contain up to 3 embeddings covering from 2 to 9 triple patterns. The residual part of the query ranges from 10 to 3 triple patterns respectively. It is also possible to observe that covering a larger number of query patterns using as few embeddings as possible (reducing in this way the residual part of the query) decreases the real processing time. This factor, in addition to the estimated cost, provides valuable knowledge to choose an optimal cover for a given query.

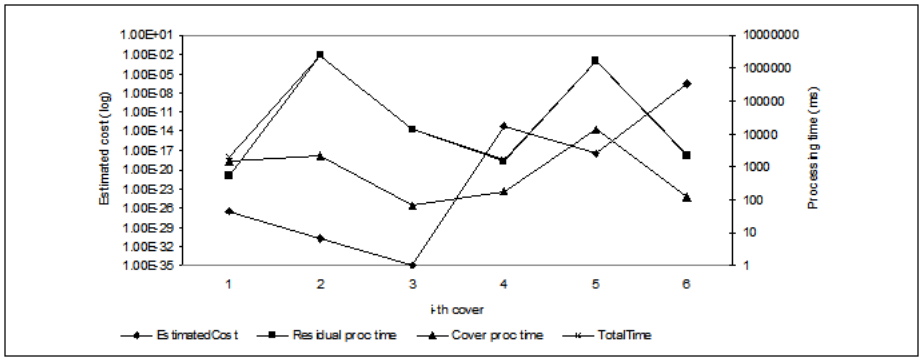



**Fig. 13.** Estimated versus real cost for Query2. Estimated costs correlate with cover real processing time however, residual processing time consumes most of the real processing time.

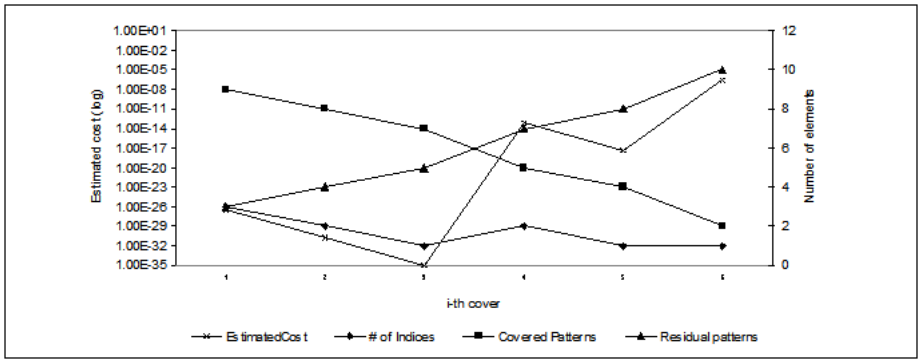



**Fig. 14.** For Query 2: Estimated cost versus number of covered patterns and number of embeddings; for explanation, see Figure 12.

### 7.4 Current Limitations

Our current approach using materialized views as indexes for SPARQL queries has a number of limitations:

- The *WHERE-CLAUSE* of a SPARQL query consists of a *Basic Graph Pattern*.
- Filters and modifiers are not considered.
- Blank nodes are not allowed.

In future work, we plan to extend our implementation, especially to allow the use of filters and modifiers. The use of multiple basic graph patterns would also provide more flexibility to create indexes.

## 8 Conclusions and future work

In this article we have proposed a novel method to speed-up the execution of SPARQL queries. We introduced a logical and physical framework to answer a SPARQL query using materialized views as indexes. At runtime, queries are analyzed to see whether they can be executed by using one or more of those precomputed views. Experiments have shown that the achievable performance gains are considerable. However, a closer look also revealed that our cost model still can be improved. This, and the removal of several technical limitations of our approach which restrict the types of queries it can handle, will be the focus of our future work.

First, we defined an RDF data dictionary to identify all different resources. Second, we preprocessed SPARQL queries materializing the result sets. Our approach indexes not only RDF data but proposes a native SPARQL index method. The use of RDF-MatView indexes minimizes query pattern comparison against the RDF data set. We analyze query and index patterns to generate all covers for a SPARQL query and also the rewriting of the latter to use indexes and get the final query result set. Even when the execution time of our approach is higher than the execution time in ARQ, the processing time significantly decreases using some strategies of our approach. It is important to notice that execution time of our approach remains constant and depends only on the size of the query pattern. Additionally, we analyzed and compared all generated covers. Our results show that besides the costs, the number of indexes and size of a cover, as well as the residual query patterns are determining to select an optimal cover. We restrict our approach to execute queries containing only a basic graph pattern. In the future we will continue working on the optimization of the query processing analyzing other storage schemas as well as on the improvement of our algorithms for an optimal generation of execution plans using more complex queries. Another interesting and promising topic to extend our approach is index selection for SPARQL queries, i.e., given a workload of SPARQL queries, perform a suggestion of which indexes should be built to efficiently answer these queries.

## References


1. Manola, F., Miller, E.: RDF Primer (February 2004) W3C Recommendation.
2. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF (April 2008) W3C Recommendation.
3. Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D.: Efficient RDF storage and retrieval in Jena2. In: Proc. First International Workshop on Semantic Web and Databases. (2003)
4. Stephen Harris, N.G.: 3store: Efficient bulk rdf storage. In: 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03). (2003)
5. Harris, S.: Sparql query processing with conventional relational database systems. In: International Workshop on Scalable Semantic Web Knowledge Base System. (2005)
6. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A generic architecture for storing and querying rdf and rdf schema. In: International Semantic Web Conference. (2002) 54–68
7. Dataset, U.R.: http://dev.isb-sib.ch/projects/uniprot-rdf/
8. Project, W.S.C.: Linking open data on the semantic web. http://esw.w3.org/topic/sweoig/taskforces/communityprojects/linkingopendata/
9. Bio2RDF. http://bio2rdf.org/ (2009)
10. Heese, R., Leser, U., Quilitz, B., Rothe, C.: Index support for sparql. European Semantic Web Conference, Innsbruck, Austria (2007)
11. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable semantic web data management using vertical partitioning. In: VLDB '07: Proceedings of the 33rd international conference on Very large data bases, VLDB Endowment (2007) 411–422
12. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. Proc. VLDB Endow. **1**(1) (2008) 1008–1019
13. Neumann, T., Weikum, G.: Rdf-3x: a risc-style engine for rdf. Proc. VLDB Endow. **1**(1) (2008) 647–659
14. Udrea, O., Pugliese, A., Subrahmanian, V.S.: Grin: A graph based rdf index. In: AAAI. (2007) 1465–1470
15. Groppe, S., Groppe, J., Linnemann, V.: Using an Index of Precomputed Joins in order to speed up SPARQL Processing. In Cardoso, J., Cordeiro, J., Filipe, J., eds.: Proceedings 9th International Conference on Enterprise Information Systems (ICEIS 2007 (1), Volume DISI), Funchal, Madeira, Portugal, INSTICC (June 12 - 16 2007) 13–20
16. Connolly, T.M., Begg, C.E., Strachan, A.D.: Database systems: a practical approach to design, implementation and management. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1996)
17. Halevy, A.Y.: Answering queries using views: A survey. The VLDB Journal **10**(4) (2001) 270–294
18. Patrick Hayes, B.M.: Rdf semantics (February 2004) W3C Recommendation.
19. ARQJena: Arq - a sparql processor for jena. http://jena.sourceforge.net/ARQ/ (2010)
20. Chebotko, A., Lu, S., Jamil, H.M., Fotouhi, F.: Semantics preserving sparql-to-sql query translation for optional graph patterns. Technical report, Department of Computer Science, Wayne State University (2006)
21. Bizer, C., Schultz, A.: The berlin sparql benchmark. International Journal On Semantic Web and Information Systems - Special Issue on Scalability and Performance of Semantic Web Systems, 2009 (2009)
22. Comer, D.: The difficulty of optimum index selection. ACM Trans. Database Syst. **3**(4) (1978) 440–445
23. Caprara, A., Fischetti, M., Maio, D.: Exact and approximate algorithms for the index selection problem in physical database design. IEEE Transactions on Knowledge and Data Engineering **7**(6) (1995) 955–967

24. Chaudhuri, S., Narasayya, V.R.: An efficient cost-driven index selection tool for microsoft sql server. In: VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1997) 146–155

# A Appendix: Queries and indexes

## A.1 Queries

All queries and indexes use the following namespaces:

```
PREFIX bsbm-inst:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

### Query 1

```
SELECT * WHERE {
    ?product rdfs:label ?label .
    ?product a ?ProductType .
    ?product bsbm:productFeature ?ProductFeature1 .
    ?product bsbm:productFeature ?ProductFeature2 .
    ?product bsbm:productPropertyNumeric1 ?value1 .
}
```

### Query 2

```
SELECT * WHERE {
    ?product rdfs:label ?label .
    ?product rdfs:comment ?comment .
    ?product bsbm:producer ?p .
    ?p rdfs:label ?producer .
    ?product dc:publisher ?p .
    ?product bsbm:productFeature ?f .
    ?f rdfs:label ?productFeature .
    ?product bsbm:productPropertyTextual1 ?propertyTextual1 .
    ?product bsbm:productPropertyTextual2 ?propertyTextual2 .
    ?product bsbm:productPropertyTextual3 ?propertyTextual3 .
    ?product bsbm:productPropertyNumeric1 ?propertyNumeric1 .
    ?product bsbm:productPropertyNumeric2 ?propertyNumeric2 .
 }
```

### Query 3

```
SELECT * WHERE {
    ?product rdfs:label ?productLabel .
    ?offer bsbm:product ?product .
    ?offer bsbm:price ?price .
    ?offer bsbm:vendor ?vendor .
    ?vendor rdfs:label ?vendorTitle .
```

```
    ?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#DE> .
    ?offer dc:publisher ?vendor .
    ?offer bsbm:validTo ?date .
    ?review bsbm:reviewFor ?product .
    ?review rev:reviewer ?reviewer .
    ?reviewer foaf:name ?revName .
    ?review dc:title ?revTitle .
    ?review bsbm:rating1 ?rating1 .
}
```

### A.2 Indexes

**Index 1**

```
SELECT  * WHERE {
    ?product  rdfs:label      ?label ;
    rdf:type        ?ProductType ;
    bsbm:productFeature     ?ProductFeature1 .
}
```

**Index 2**

```
SELECT * WHERE {
    ?product a ?ProductType .
    bsbm:productPropertyNumeric1 ?value1 .
}
```

**Index 3**

```
SELECT * WHERE {
    ?product bsbm:producer ?p .
    ?p rdfs:label ?producer .
    ?product dc:publisher ?p .
    ?product bsbm:productFeature ?f .
 }
```

**Index 4**

```
SELECT * WHERE {
    ?product bsbm:productFeature ?f .
    ?f rdfs:label ?productFeature .
 }
```

**Index 5**

```
SELECT * WHERE {
    ?product rdfs:label ?label .
    ?product rdfs:comment ?comment .
    ?product bsbm:producer ?p .
    ?p rdfs:label ?producer .
    ?product dc:publisher ?p .
    ?product bsbm:productPropertyTextual1 ?propertyTextual1 .
    ?product bsbm:productPropertyNumeric1 ?propertyNumeric1 .
}
```

**Index 6**

```
SELECT * WHERE {
    ?product rdfs:label ?productLabel .
    ?offer bsbm:product ?product .
    ?offer bsbm:price ?price .
    ?offer bsbm:vendor ?vendor .
}
```

**Index 7**

```
SELECT * WHERE {
    ?product rdfs:label ?productLabel .
    ?review bsbm:reviewFor ?product .
    ?review rev:reviewer ?reviewer .
    ?reviewer foaf:name ?revName .
    ?review dc:title ?revTitle .
}
```

**Index 8**

```
SELECT * WHERE {
    ?offer bsbm:product ?product .
    ?offer bsbm:price ?price .
    ?offer bsbm:vendor ?vendor .
    ?vendor rdfs:label ?vendorTitle .
    ?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#DE> .
    ?offer dc:publisher ?vendor .
    ?review bsbm:reviewFor ?product .
    ?review rev:reviewer ?reviewer .
    ?reviewer foaf:name ?revName .
}
```

**Index 9**

```
SELECT * WHERE {
    ?review bsbm:reviewFor ?product .
    ?review dc:title ?title .
    ?review rev:text ?text .
}
```

**Index 10**

```
SELECT * WHERE {
    ?review bsbm:reviewFor ?product .
    ?review bsbm:rating1 ?rating1 .
}
```